



# 智能合约安全审计报告





审计报告编号: 202103192204

报告查询名称: 0xDex

审计项目信息:

审计项目哈希 (SHA256)	d19c747c2bd7b7509cbbaa4860c633c0829c11eca54cdf3aac88ece165bd8b08d
-----------------	---

审计项目详细条目:

合约名称	合约所在路径
StakingRewards	staking\src\StakingRewards.sol
0xDexERC20	oxdex-contract\contracts\0xDexERC20.sol
0xDexFactory	oxdex-contract\contracts\0xDexFactory.sol
0xDexPair	oxdex-contract\contracts\0xDexPair.sol
0xDexRouter	oxdex-contract\contracts\0xDexRouter.sol

合约审计开始日期: 2021. 03. 16

合约审计完成日期: 2021. 03. 19

审计结果: 通过

审计团队: 成都链安科技有限公司

审计类型及结果:

序号	审计类型	审计子项	审计结果
1	代码规范审计	编译器版本安全审计	通过
		弃用项审计	通过
		冗余代码审计	通过
		SafeMath 功能审计	通过
		require/assert 使用审计	通过
		gas 消耗审计	通过
		可见性规范审计	通过
		fallback 函数使用审计	通过

2	通用漏洞审计	整型溢出审计	通过
		重入攻击审计	通过
		伪随机数生成审计	通过
		交易顺序依赖审计	通过
		拒绝服务攻击审计	通过
		函数调用权限审计	通过
		call/delegatecall 安全审计	通过
		返回值安全审计	通过
		tx.origin 使用安全审计	通过
		重放攻击审计	通过
3	业务审计	变量覆盖审计	通过
		业务逻辑审计	通过
		业务实现审计	通过

备注：审计意见及建议请见代码注释。

免责声明：本次审计仅针对本报告载明的审计类型及结果表中给定的审计类型范围进行审计，其他未知安全漏洞不在本次审计责任范围之内。成都链安科技仅根据本报告出具前已经存在或发生的攻击或漏洞出具本报告，对于出具以后存在或发生的新的攻击或漏洞，成都链安科技无法判断其对智能合约安全状况可能的影响，亦不对此承担责任。本报告所作的安全审计分析及其他内容，仅基于合约提供者在本报告出具前已向成都链安科技提供的文件和资料，且该部分文件和资料不存在任何缺失、被篡改、删减或隐瞒的前提下作出的；如提供的文件和资料存在信息缺失、被篡改、删减、隐瞒或反映的情况与实际情况不符等情况或提供文件和资料在本报告出具后发生任何变动的，成都链安科技对由此而导致的损失和不利影响不承担任何责任。成都链安科技出具的本审计报告系根据合约提供者提供的文件和资料依靠成都链安科技现掌握的技术而作出的，由于任何机构均存在技术的局限性，成都链安科技作出的本审计报告仍存在无法完整检测出全部风险的可能性，成都链安科技对由此产生的损失不承担任何责任。

本声明最终解释权归成都链安科技所有。

## 审计结果说明：

本公司采用形式化验证、静态分析、动态分析、典型案例测试和人工审核的方式对0xDex项目智能合约的代码规范性、安全性以及业务逻辑三个方面进行多维度全面的安全审计。经审计，0xDex项目的智能合约通过所有检测项，合约审计结果为通过。以下为本项目的合约详细审计信息。



## 1 代码规范审计

### 1.1 编译器版本安全审计

老版本的编译器可能会导致各种已知的安全问题，建议开发者在代码中指定合约代码采用最新的编译器版本，并消除编译器告警。

使用合约指定版本的编译器编译合约，无任何编译器警告。

- 安全建议：无
- 审计结果：通过

### 1.2 弃用项审计

Solidity智能合约开发语言处于快速迭代中，部分关键字已被新版本的编译器弃用，如throw、years等，为了消除其可能导致的隐患，合约开发者不应该使用当前编译器版本已弃用的关键字。

- 安全建议：无
- 审计结果：通过

### 1.3 冗余代码审计

智能合约中的冗余代码会降低代码可读性，并可能需要消耗更多的gas用于合约部署，建议消除冗余代码。

- 安全建议：无
- 审计结果：通过

### 1.4 SafeMath功能审计

检查合约中是否正确使用SafeMath库内的函数进行数学运算，或者进行其他防溢出的检查。

- 安全建议：无
- 审计结果：通过

### 1.5 require/assert使用审计

Solidity使用状态恢复异常来处理错误。这种机制将会撤消对当前调用(及其所有子调用)中的状态所做的所有更改，并向调用者标记错误。函数assert和require可用于检查条件并在条件不满足时抛出异常。assert函数只能用于测试内部错误，并检查非变量。require函数用于确认条件有效性，例如输入变量，或合约状态变量是否满足条件，或验证外部合约调用的返回值。

- 安全建议：无
- 审计结果：通过

### 1.6 gas消耗审计

以太坊主链的虚拟机执行合约代码需要消耗gas，当gas不足时，代码执行会抛出out of gas异常，并撤销所有状态变更。合约开发者需要控制代码的gas消耗，避免因为gas不足导致函数执行一直失败。

- 安全建议：无
- 审计结果：通过

### 1.7 可见性规范审计

检查合约函数的可见性是否符合设计要求。

- 安全建议：无
- 审计结果：通过

### 1.8 fallback函数使用审计

检查在当前合约中是否正确使用了fallback函数。

本项目中的Router合约指定了本合约可接收来自WETH合约的ETH转账。

```
28 |     receive() external payable {  
29 |         assert(msg.sender == WETH); // only accept ETH via fallback from the WETH contract  
30 |     }  
31 | }
```

图 1 fallback函数源码

- 安全建议：无
- 审计结果：通过

## 2 通用漏洞审计

### 2.1 整型溢出审计

整型溢出是很多语言都存在的安全问题，它们在智能合约中尤其危险。Solidity最多能处理256位的数字( $2^{256}-1$ )，最大数字增加1会溢出得到0。同样，当数字为uint类型时，0减去1会下溢得到最大数字值。溢出情况会导致不正确的结果，特别是如果其可能的结果未被预期，可能会影响程序的可靠性和安全性。

- 安全建议：无
- 审计结果：通过

### 2.2 重入攻击审计

重入漏洞是最典型的智能合约漏洞，曾导致了The DAO被攻击。该漏洞原因是Solidity中的call.value()函数在被用来发送ETH的时候会消耗它接收到的所有gas，当调用call.value()函数发送ETH的逻辑顺序存在错误时，就会存在重入攻击的风险。

- 安全建议：无
- 审计结果：通过

### 2.3 伪随机数生成审计



智能合约中可能会使用到随机数，在solidity下常见的是用block区块信息作为随机因子生成，但是这样使用是不安全的，区块信息是可以被矿工控制或被攻击者在交易时获取到，这类随机数在一定程度上是可预测或可碰撞的，比较典型的例子就是fomo3d的airdrop随机数可以被碰撞。

➤ 安全建议：无

➤ 审计结果：通过

## 2.4 交易顺序依赖审计

在交易打包执行过程中，面对相同难度的交易时，矿工往往会选择gas费用高的优先打包，因此用户可以指定更高的gas费用，使自己的交易优先被打包执行。

➤ 安全建议：无

➤ 审计结果：通过

## 2.5 拒绝服务攻击审计

拒绝服务攻击，即Denial of Service，可以使目标无法提供正常的服务。在智能合约中也会存在此类问题，由于智能合约的不可更改性，该类攻击可能使得合约永远无法恢复正常工作状态。导致智能合约拒绝服务的原因有很多，包括在作为交易接收方时的恶意revert、代码设计缺陷导致gas耗尽等等。

➤ 安全建议：无

➤ 审计结果：通过

## 2.6 函数调用权限审计

智能合约如果存在高权限功能，如：铸币、自毁、change owner等，需要对函数调用做权限限制，避免权限泄露导致的安全问题。

➤ 安全建议：无

➤ 审计结果：通过

## 2.7 call/delegatecall安全审计

Solidity中提供了call/delegatecall函数来进行函数调用，如果使用不当，会造成call注入漏洞，例如call的参数如果可控，则可以控制本合约进行越权操作或调用其他合约的危险函数。

➤ 安全建议：无

➤ 审计结果：通过

## 2.8 返回值安全审计

在Solidity中存在transfer()、send()、call.value()等方法中，transfer转账失败交易会回滚，而send和call.value转账失败会return false，如果未对返回做正确判断，则可能会执行到未预期的逻辑；另外在ERC20 Token的transfer/transferFrom功能实现中，也要避免转账失败return false的情况，以免造成假充值漏洞。

- 安全建议：无
- 审计结果：通过

## 2.9 tx.origin使用安全审计

在智能合约的复杂调用中，tx.origin表示交易的初始创建者地址，如果使用tx.origin进行权限判断，可能会出现错误；另外，如果合约需要判断调用方是否为合约地址时则需要使用tx.origin，不能使用extcodesize。

- 安全建议：无
- 审计结果：通过

## 2.10 重放攻击审计

重放攻击是指如果两份合约使用了相同的代码实现，并且身份鉴权在传参中，当用户在向一份合约中执行一笔交易，交易信息可以被复制并且向另一份合约重放执行该笔交易。

本项目的LP代币可接收签名授权，如下图，对应函数中要求的签名数据必须包含对应地址nonce值，且合约中记录的地址nonce递增，可避免签名数据被重复利用。

```
93     function permit(
94         address owner,
95         address spender,
96         uint256 value,
97         uint256 deadline,
98         uint8 v,
99         bytes32 r,
100        bytes32 s
101    ) external {
102        require(deadline >= block.timestamp, '0xDex: EXPIRED');
103        bytes32 digest =
104            keccak256(
105                abi.encodePacked(
106                    '\x19\x01',
107                    DOMAIN_SEPARATOR,
108                    keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[owner]++, deadline))
109                )
110            );
111        address recoveredAddress = ecrecover(digest, v, r, s);
112        require(recoveredAddress != address(0) && recoveredAddress == owner, '0xDex: INVALID_SIGNATURE');
113        _approve(owner, spender, value);
114    }
115 }
```

图 2 permit函数源码

- 安全建议：无
- 审计结果：通过

## 2.11 变量覆盖审计

智能合约中存在着复杂的变量类型，例如结构体、动态数组等，如果使用不当，对其赋值后，可能导致覆盖已有状态变量的值，造成合约执行逻辑异常。

- 安全建议：无
- 审计结果：通过



### 3 业务审计

#### 3.1 StakingRewards合约

##### 3.1.1 合约管理权限控制

➤ 业务描述:

合约管理员owner(默认是合约部署者)可调用nominateNewOwner函数设置指定地址为提名owner; 并且提名owner的地址需要调用acceptOwnership函数接收owner权限。

➤ 相关函数: nominateNewOwner、acceptOwnership

➤ 安全建议: 无

➤ 审计结果: 通过

##### 3.1.2 合约暂停管理

➤ 业务描述:

合约管理员owner(默认是合约部署者)可调用setPaused函数设置合约的暂停状态。

➤ 相关函数: setPaused

➤ 安全建议: 无

➤ 审计结果: 通过

##### 3.1.3 设置奖励分配地址

➤ 业务描述:

合约管理员owner(默认是合约部署者)可调用setRewardsDistribution函数设置指定地址为奖励分配地址。

➤ 相关函数: setRewardsDistribution

➤ 安全建议: 无

➤ 审计结果: 通过

##### 3.1.4 分配周期奖励

➤ 业务描述:

具有奖励分配权限的地址可调用notifyRewardAmount函数分配周期奖励。如下图所示, 该函数会触发updateReward修饰器更新系统相关的质押参数, 如果当前周期已经结束, 会以输入参数reward为周期奖励开启下一个周期; 如果当前周期未结束, 则会将输入参数reward对应的代币添加至未发放的奖励, 并开启新的周期。

```

130   function notifyRewardAmount(uint256 reward) external override onlyRewardsDistribution updateReward(address(0)) {
131     if (block.timestamp >= periodFinish) {
132       rewardRate = reward.div(rewardsDuration);
133     } else {
134       uint256 remaining = periodFinish.sub(block.timestamp);
135       uint256 leftover = remaining.mul(rewardRate);
136       rewardRate = reward.add(leftover).div(rewardsDuration);
137     }
138
139     // Ensure the provided reward amount is not more than the balance in the contract.
140     // This keeps the reward rate in the right range, preventing overflows due to
141     // very high values of rewardRate in the earned and rewardsPerToken functions;
142     // Reward + leftover must be less than 2^256 / 10^18 to avoid overflow.
143     uint balance = rewardsToken.balanceOf(address(this));
144     require(rewardRate <= balance.div(rewardsDuration), "Provided reward too high");
145
146     lastUpdateTime = block.timestamp;
147     periodFinish = block.timestamp.add(rewardsDuration);
148     emit RewardAdded(reward);
149   }
  
```

图 3 notifyRewardAmount 函数源码

➤ 相关函数: notifyRewardAmount

➤ 安全建议: 无

➤ 审计结果: 通过

### 3.1.5 设置周期奖励相关参数

➤ 业务描述:

合约的管理员 owner 可调用 updatePeriodFinish 函数来设置本周期结束时间 periodFinish。

```

151   // End rewards commission controller
152   function updatePeriodFinish(uint timestamp) external onlyOwner updateReward(address(0)) {
153     periodFinish = timestamp;
154   }
  
```

图 4 updatePeriodFinish 函数源码

**需要注意:** 如果管理员误操作将 periodFinish 设置为比上一次更新时间 (lastUpdateTime) 小，那么将会导致整个合约产生异常而无法使用 (且不可修复)。因为合约的关键操作都会调用 updateReward 修饰器去更新合约状态，其内部会调用 rewardPerToken 函数更新当前代币的累计价值，而在 rewardPerToken 函数中，会计算 lastTimeRewardApplicable 函数值距离上一次更新时间的累计奖励，而如果 periodFinish 较小，就会出现 lastTimeRewardApplicable 函数值小于 periodFinish 的情况，导致出现图 6 中红框部分的 sub 出现异常，对应的函数调用失败。

```

175
174   modifier updateReward(address account) {
175     rewardPerTokenStored = rewardPerToken();
176     lastUpdateTime = lastTimeRewardApplicable();
177     if (account != address(0)) {
178       rewards[account] = earned(account);
179       userRewardPerTokenPaid[account] = rewardPerTokenStored;
180     }
181   }
182 }
  
```

图 5 updateReward 修饰器源码



```
63     function lastTimeRewardApplicable() public override view returns (uint256) {
64         return Math.min(block.timestamp, periodFinish);
65     }
66
67     function rewardPerToken() public override view returns (uint256) {
68         if (_totalSupply == 0) {
69             return rewardPerTokenStored;
70         }
71         return
72             rewardPerTokenStored.add(
73                 lastTimeRewardApplicable().sub(lastUpdateTime).mul(rewardRate).mul(1e18).div(_totalSupply)
74             );
75     }
```

图 6 lastTimeRewardApplicable 函数和 rewardPerToken 函数源码

另外，合约管理员 owner 可调用 setRewardsDuration 函数设置周期时长，该操作下的实际奖励会在下个周期生效(奖励分配管理员调用 notifyRewardAmount 函数)，但会影响 rewardsDuration 和 getRewardForDuration 函数的查询值。

- **相关函数:** updatePeriodFinish、updateReward(修饰器)、lastTimeRewardApplicable、rewardPerToken、setRewardsDuration、getRewardForDuration
- **安全建议:** 删除updatePeriodFinish函数。
- **修复结果:** 忽略。经项目方确认，updatePeriodFinish函数是为了满足项目需求，且承诺管理员会谨慎操作。
- **审计结果:** 通过

### 3.1.6 提取其他ERC20代币

- **业务描述:**

合约管理员owner可调用recoverERC20函数提取本合约持有的ERC20代币，函数中限制了提取的代币不能是质押代币，但并没限制提取的代币不能是奖励代币。虽然按照合约逻辑，奖励代币由手动发送至本合约，可能存在发送过量的情况，需要本函数提取，但仍建议在此函数限制提取代币的不能是奖励代币，或直接删除此函数，以免影响质押用户领取质押奖励。

```
157     function recoverERC20(address tokenAddress, uint256 tokenAmount) external onlyOwner {
158         require(tokenAddress != address(stakingToken), "Cannot withdraw the staking token");
159         IERC20(tokenAddress).safeTransfer(owner, tokenAmount);
160         emit Recovered(tokenAddress, tokenAmount);
161     }
```

图 7 recoverERC20 函数源码

- **相关函数:** recoverERC20
- **安全建议:** 删除recoverERC20函数，或者限制提取的代币不能是奖励代币；另外建议在构造函数中添加质押代币和奖励代币不能为同一代币的代码限制逻辑。
- **修复结果:** 忽略。经项目方确认，忽略该问题，该函数的存在是符合项目设计需求。
- **审计结果:** 通过

### 3.1.7 质押代币

- **业务描述:**



本合约提供了 stake 和 stakeWithPermit 两个函数供用户进行质押。

其中，用户可调用 stakeWithPermit 函数并传入对应的代币授权签名数据进行代币质押。如下图，函数会先更新用户的质押数据；然后调用对应质押合约的 permit 函数，并传入签名参数向本合约授权；最后调用质押合约的 safeTransferFrom 函数转账质押代币至本合约，完成质押。

```
87     function stakeWithPermit(uint256 amount, uint deadline, uint8 v, bytes32 r, bytes32 s) external nonReentrant updateReward(msg.sender) {
88         require(amount > 0, "Cannot stake 0");
89         _totalSupply = _totalSupply.add(amount);
90         _balances[msg.sender] = _balances[msg.sender].add(amount);
91
92         EIP2612(address(stakingToken)).permit(msg.sender, address(this), amount, deadline, v, r, s);
93
94         stakingToken.safeTransferFrom(msg.sender, address(this), amount);
95         emit Staked(msg.sender, amount);
96     }
```

图 8 stakeWithPermit 函数源码

而 stake 函数则需要调用者提前对本合约授权，然后调用本合约的 stake 函数进行质押。

```
98     function stake(uint256 amount) external override nonReentrant notPaused updateReward(msg.sender) {
99         require(amount > 0, "Cannot stake 0");
100        _totalSupply = _totalSupply.add(amount);
101        _balances[msg.sender] = _balances[msg.sender].add(amount);
102        stakingToken.safeTransferFrom(msg.sender, address(this), amount);
103        emit Staked(msg.sender, amount);
104    }
```

图 9 stake 函数源码

**需要注意：**其中，stake 函数添加了 notPaused 修饰器，表示如果合约暂停，用户将不能通过此函数进行质押；但另一个质押函数 stakeWithPermit 却没有暂停的限制。即使合约处于暂停状态，也可通过 stakeWithPermit 函数进行质押。

```
86
87     function stakeWithPermit(uint256 amount, uint deadline, uint8 v, bytes32 r, bytes32 s) external nonReentrant updateReward(msg.sender) {
88         require(amount > 0, "Cannot stake 0");
89         _totalSupply = _totalSupply.add(amount);
90         _balances[msg.sender] = _balances[msg.sender].add(amount);
91
92         EIP2612(address(stakingToken)).permit(msg.sender, address(this), amount, deadline, v, r, s);
93
94         stakingToken.safeTransferFrom(msg.sender, address(this), amount);
95         emit Staked(msg.sender, amount);
96     }
97
98     function stake(uint256 amount) external override nonReentrant notPaused updateReward(msg.sender) {
99         require(amount > 0, "Cannot stake 0");
100        _totalSupply = _totalSupply.add(amount);
101        _balances[msg.sender] = _balances[msg.sender].add(amount);
102        stakingToken.safeTransferFrom(msg.sender, address(this), amount);
103        emit Staked(msg.sender, amount);
104    }
```

图 10 合约质押代币相关函数

- **相关函数：**stake、stakeWithPermit
- **安全建议：**建议在 stakeWithPermit 函数声明中添加 notPaused 修饰器
- **修复结果：**忽略
- **审计结果：**通过

### 3.1.8 提取质押代币



➤ 业务描述:

已质押用户可调用 withdraw 函数提取指定数量的质押代币，如下图所示，函数会更新用户的质押信息，并将对应数量的质押代币发送至调用者地址。

```
106     function withdraw(uint256 amount) public override nonReentrant updateReward(msg.sender) {
107         require(amount > 0, "Cannot withdraw 0");
108         _totalSupply = _totalSupply.sub(amount);
109         _balances[msg.sender] = _balances[msg.sender].sub(amount);
110         stakingToken.safeTransfer(msg.sender, amount);
111         emit Withdrawn(msg.sender, amount);
112     }
```

图 11 withdraw 函数源码

另外，已质押用户也可调用 exit 函数退出质押，该过程也会提取出调用者的全部质押代币。

➤ 相关函数: withdraw、exit

➤ 安全建议: 无

➤ 审计结果: 通过

### 3.1.9 领取质押奖励

➤ 业务描述:

用户可调用 getReward 函数领取质押奖励，也可调用 exit 函数领取。

```
113
114     function getReward() public override nonReentrant updateReward(msg.sender) {
115         uint256 reward = rewards[msg.sender];
116         if (reward > 0) {
117             rewards[msg.sender] = 0;
118             rewardsToken.safeTransfer(msg.sender, reward);
119             emit RewardPaid(msg.sender, reward);
120         }
121     }
```

图 12 getReward 函数源码

➤ 相关函数: getReward、exit

➤ 安全建议: 无

➤ 审计结果: 通过

### 3.2 0xDexERC20合约

#### 3.2.1 代币基本信息

➤ 业务描述:

本合约实现了基本的ERC20代币，用于作为流动性的凭证，代币详细情况如下：



Token name	0xDex LP
Token symbol	0XLP
decimals	18
totalSupply	初始为0(可铸币, 可销毁)
Token type	ERC20

表 1 代币基本信息

- 安全建议：无
- 审计结果：通过

### 3.2.2 代币基本功能

- 业务描述：

本合约代币严格遵循 ERC20 Token 标准，实现了标准中要求的转账、授权和代理转账等功能。但需要注意：用户在使用 approve 函数修改授权值时，可能出现多重授权；建议先将当前授权值置为 0，再进行新的授权。

- 相关函数：name、symbol、decimals、transfer、approve、transferFrom、balanceOf、allowance、totalSupply
- 安全建议：无
- 审计结果：通过

### 3.2.3 签名授权

- 业务描述：

如下图所示，用户可调用 permit 函数传入授权参数和签名参数完成 owner 对 spender 地址的授权。该函数中检查了对应签名者地址和对应 nonce。

```
92     function permit(
93         address owner,
94         address spender,
95         uint256 value,
96         uint256 deadline,
97         uint8 v,
98         bytes32 r,
99         bytes32 s
100    ) external {
101        require(deadline >= block.timestamp, '0xDex: EXPIRED');
102        bytes32 digest =
103            keccak256(
104                abi.encodePacked(
105                    '\x19\x01',
106                    DOMAIN_SEPARATOR,
107                    keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[owner]++, deadline))
108                )
109            );
110        address recoveredAddress = ecrecover(digest, v, r, s);
111        require(recoveredAddress != address(0) && recoveredAddress == owner, '0xDex: INVALID_SIGNATURE');
112        _approve(owner, spender, value);
113    }
114 }
```

图 13 permit函数源码

- 相关函数: permit
- 安全建议: 无
- 审计结果: 通过

### 3.3 0xDexFactory合约

#### 3.3.1 费用接收地址管理

- 业务描述:

该合约在部署时，需要指定地址作为项目的费用接收地址管理员，费用接收地址管理员可调用 setFeeToSetter 函数更新管理员地址；调用 setFeeTo 函数设置费用接收地址。

- 相关函数: setFeeToSetter、setFeeTo
- 安全建议: 无
- 审计结果: 通过

#### 3.3.2 创建pair合约

- 业务描述:

任意用户可调用本合约的 createPair 函数创建指定地址间的交易对，如下图所示，函数首先要求两个地址不能为同一地址，且根据地址大小进行排序(升序)，token0 为较小的地址，但不能为零地址；然后根据交易对模板合约 0xDexPair 的字节码和对应地址的 salt 创建交易对合约；最后调用交易对合约的 initialize 函数对交易对合约进行初始化。

```

23   function createPair(address tokenA, address tokenB) external returns (address pair) {
24     require(tokenA != tokenB, '0xDex: IDENTICAL_ADDRESSES');
25     (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
26     require(token0 != address(0), '0xDex: ZERO_ADDRESS');
27     require(getPair[token0][token1] == address(0), '0xDex: PAIR_EXISTS'); // single check is sufficient
28     bytes memory bytecode = type(0xDexPair).creationCode;
29     bytes32 salt = keccak256(abi.encodePacked(token0, token1));
30     assembly {
31       pair := create2(0, add(bytecode, 32), mload(bytecode), salt)
32     }
33     I0xDexPair(pair).initialize(token0, token1);
34     getPair[token0][token1] = pair;
35     getPair[token1][token0] = pair; // populate mapping in the reverse direction
36     allPairs.push(pair);
37     emit PairCreated(token0, token1, pair, allPairs.length);
38   }
  
```

图 14 createPair 函数源码

➤ 相关函数: createPair、initialize

➤ 安全建议: 无

➤ 审计结果: 通过

### 3.4 0xDexPair合约

#### 3.4.1 合约初始化

➤ 业务描述:

该合约在创建后，会调用本合约的 initialize 函数对合约进行初始化，如下图， initialize 函数要求调用者必须 factory 地址，并设置了交易对的两个代币地址。

```

78   function initialize(address _token0, address _token1) external {
79     require(msg.sender == factory, '0xDex: FORBIDDEN'); // sufficient check
80     token0 = _token0;
81     token1 = _token1;
82   }
  
```

图 15 initialize 函数源码

➤ 相关函数: initialize

➤ 安全建议: 无

➤ 审计结果: 通过

### 3.4.2 添加流动性

➤ 业务描述:

用户在向本合约发送对应的两种代币后(交易对所要求的代币)，需调用本合约的 mint 函数添加指定地址的流动性。如下图，函数获取合约当前的代币持有量后，会调用 \_mintFee 函数发放费用；然后根据当前的 LP 代币总量，选择是否需要扣除系统所需的最小流动性数值；接着，调用 \_mint 函数发放用户抵押的 LP 代币、更新合约的代币存储量。



```
127     function mint(address to) external lock returns (uint256 liquidity) {
128         (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
129         uint256 balance0 = IERC20(token0).balanceOf(address(this));
130         uint256 balance1 = IERC20(token1).balanceOf(address(this));
131         uint256 amount0 = balance0.sub(_reserve0);
132         uint256 amount1 = balance1.sub(_reserve1);
133
134         bool feeOn = _mintFee(_reserve0, _reserve1);
135         uint256 _totalSupply = totalSupply; // gas savings, must be defined here since totalSupply can update in _mintFee
136         if (_totalSupply == 0) {
137             liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
138             _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens
139         } else {
140             liquidity = Math.min(amount0.mul(_totalSupply) / _reserve0, amount1.mul(_totalSupply) / _reserve1);
141         }
142         require(liquidity > 0, '0xDex: INSUFFICIENT_LIQUIDITY_MINTED');
143         _mint(to, liquidity);
144
145         _update(balance0, balance1, _reserve0, _reserve1);
146         if (feeOn) kLast = uint256(_reserve0).mul(_reserve1); // reserve0 and reserve1 are up-to-date
147         emit Mint(msg.sender, amount0, amount1);
148     }
```

图 16 mint 函数源码

\_mintFee 函数会判断是否开启手续费收取，如果已开启，则会根据当前与上一次 K 值的差值发放费用。

```
105     // if fee is on, mint liquidity equivalent to 1/6th of the growth in sqrt(k)
106     function _mintFee(uint112 _reserve0, uint112 _reserve1) private returns (bool feeOn) {
107         address feeTo = IOxDexFactory(factory).feeTo();
108         feeOn = feeTo != address(0);
109         uint256 _kLast = kLast; // gas savings
110         if (feeOn) {
111             if (_kLast != 0) {
112                 uint256 rootK = Math.sqrt(uint256(_reserve0).mul(_reserve1));
113                 uint256 rootKLast = Math.sqrt(_kLast);
114                 if (rootK > rootKLast) {
115                     uint256 numerator = totalSupply.mul(rootK.sub(rootKLast));
116                     uint256 denominator = rootK.mul(5).add(rootKLast);
117                     uint256 liquidity = numerator / denominator;
118                     if (liquidity > 0) _mint(feeTo, liquidity);
119                 }
120             }
121         } else if (_kLast != 0) {
122             kLast = 0;
123         }
124     }
```

图 17 \_mintFee 函数源码

\_update 函数会根据合约当前的实际代币持有量来更新本次交易对代币存储量，其中，如果本次更新的时间与上一次更新时间不为同一时间，则还需要更新代币累计价格。

```

85   // update reserves array on the first call per block price accumulation
86   function _update(
87     uint256 balance0,
88     uint256 balance1,
89     uint112 _reserve0,
90     uint112 _reserve1
91   ) private {
92     require(balance0 <= uint112(-1) && balance1 <= uint112(-1), 'OxDex: OVERFLOW');
93     uint32 blockTimestamp = uint32(block.timestamp % 2**32);
94     uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired
95     if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
96       // * never overflows, and + overflow is desired
97       price0CumulativeLast += uint256(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) * timeElapsed;
98       price1CumulativeLast += uint256(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) * timeElapsed;
99     }
100    reserve0 = uint112(balance0);
101    reserve1 = uint112(balance1);
102    blockTimestampLast = blockTimestamp;
103    emit Sync(reserve0, reserve1);
  }

```

图 18 \_update 函数源码

- 相关函数: mint、\_mintFee、\_update
- 安全建议: 无
- 审计结果: 通过

### 3.4.3 减少流动性

- 业务描述:

用户可将 LP 代币发送至本合约，然后调用合约的 burn 函数提取对应的代币至指定地址。如下图，函数同样会调用\_mintFee 函数发放手续费，调用\_update 函数更新合约代币的存储量，此外，函数还会根据合约持有的 LP 代币数量计算出对应可提取的代币并发送至指定的 to 地址。

```

151   function burn(address to) external lock returns (uint256 amount0, uint256 amount1) {
152     (uint112 _reserve0, uint112 _reserve1, ) = getReserves(); // gas savings
153     address _token0 = token0; // gas savings
154     address _token1 = token1; // gas savings
155     uint256 balance0 = IERC20(_token0).balanceOf(address(this));
156     uint256 balance1 = IERC20(_token1).balanceOf(address(this));
157     uint256 liquidity = balanceOf[address(this)];
158
159     bool feeOn = _mintFee(_reserve0, _reserve1);
160     uint256 _totalSupply = totalSupply; // gas savings, must be defined here since totalSupply can update in _mintFee
161     amount0 = liquidity.mul(balance0) / _totalSupply; // using balances ensures pro-rata distribution
162     amount1 = liquidity.mul(balance1) / _totalSupply; // using balances ensures pro-rata distribution
163     require(amount0 > 0 && amount1 > 0, 'OxDex: INSUFFICIENT_LIQUIDITY_BURNED');
164     _burn(address(this), liquidity);
165     _safeTransfer(_token0, to, amount0);
166     _safeTransfer(_token1, to, amount1);
167     balance0 = IERC20(_token0).balanceOf(address(this));
168     balance1 = IERC20(_token1).balanceOf(address(this));
169
170     _update(balance0, balance1, _reserve0, _reserve1);
171     if (feeOn) kLast = uint256(reserve0).mul(reserve1); // reserve0 and reserve1 are up-to-date
172     emit Burn(msg.sender, amount0, amount1, to);
  }

```

图 19 burn 函数源码

- 相关函数: burn、\_mintFee、\_update

- 安全建议：无
- 审计结果：通过

### 3.4.4 代币兑换

- 业务描述：

持有本合约指定代币的用户可调用 swap 函数进行代币兑换，如下图，函数首选会进行参数检查；然后进行代币转账(欲兑换出的代币)，并要求转账后的合约的代币存储量之积大于转账前(收取千分之三的手续费)；最后调用 \_update 更新合约的代币存储量。

```

176   function swap(
177     uint256 amount0Out,
178     uint256 amount1Out,
179     address to,
180     bytes calldata data
181   ) external lock {
182     require(amount0Out > 0 || amount1Out > 0, '0xDex: INSUFFICIENT_OUTPUT_AMOUNT');
183     (uint112 _reserve0, uint112 _reserve1, ) = getReserves(); // gas savings
184     require(amount0Out < _reserve0 && amount1Out < _reserve1, '0xDex: INSUFFICIENT_LIQUIDITY');
185
186     uint256 balance0;
187     uint256 balance1;
188   {
189     // scope for _token{0,1}, avoids stack too deep errors
190     address _token0 = token0;
191     address _token1 = token1;
192     require(to != _token0 && to != _token1, '0xDex: INVALID_TO');
193     if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically transfer tokens
194     if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically transfer tokens
195     if (data.length > 0) IOxDexCallee(to).oxdexCall(msg.sender, amount0Out, amount1Out, data);
196     balance0 = IERC20(_token0).balanceOf(address(this));
197     balance1 = IERC20(_token1).balanceOf(address(this));
198   }
199   uint256 amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) : 0;
200   uint256 amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) : 0;
201   require(amount0In > 0 || amount1In > 0, '0xDex: INSUFFICIENT_INPUT_AMOUNT');
202   {
203     // scope for reserve{0,1}Adjusted, avoids stack too deep errors
204     uint256 balance0Adjusted = balance0.mul(1000).sub(amount0In.mul(3));
205     uint256 balance1Adjusted = balance1.mul(1000).sub(amount1In.mul(3));
206     require(
207       balance0Adjusted.mul(balance1Adjusted) >= uint256(_reserve0).mul(_reserve1).mul(1000**2),
208       '0xDex: K'
209     );
210   }
211
212   _update(balance0, balance1, _reserve0, _reserve1);
213   emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
214 }
```

图 20 swap 函数源码

- 相关函数：swap、\_update

- 安全建议：无

- 审计结果：通过

### 3.4.5 取出多余的代币

- 业务描述：



任意用户可调用合约 skim 将合约多余的代币发送至指定地址。

- 相关函数: skim
- 安全建议: 无
- 审计结果: 通过

### 3.4.6 手动更新合约的代币储存量

- 业务描述:

任意用户可调用合约的 sync 函数以合约当前的实际代币持有量更新储存量(记录值)。

```
225 |     function sync() external lock {  
226 |         _update(IERC20(token0).balanceOf(address(this)), IERC20(token1).balanceOf(address(this)), reserve0, reserve1);  
227 |     }
```

图 21 sync 函数源码

- 相关函数: sync

- 安全建议: 无

- 审计结果: 通过

## 3.5 0xDexRouter合约

### 3.5.1 添加流动性

- 业务描述:

持有代币的用户可调用 addLiquidity、addLiquidityETH 向指定的交易对添加流动性。下图是 addLiquidityETH 的源码，函数会先调用 \_addLiquidity 函数计算出本次调用所需的代币数量和 ETH 数量；然后将对应的代币发送至交易对，并根据需要的 ETH 兑换成相应的 WETH 发送至对应交易对；接着调用交易对合约的 mint 函数增加调用者的流动性；最后，将多余 ETH 返回至调用者。

```
89     function addLiquidityETH(
90         address token,
91         uint256 amountTokenDesired,
92         uint256 amountTokenMin,
93         uint256 amountETHMin,
94         address to,
95         uint256 deadline
96     ) external payable virtual override ensure(deadline) returns (
97         uint256 amountToken,
98         uint256 amountETH,
99         uint256 liquidity
100    ) {
101        (amountToken, amountETH) = _addLiquidity(
102            token,
103            WETH,
104            amountTokenDesired,
105            msg.value,
106            amountTokenMin,
107            amountETHMin
108        );
109        address pair = OxDexLibrary.pairFor(factory, token, WETH);
110        TransferHelper.safeTransferFrom(token, msg.sender, pair, amountToken);
111        IWETH(WETH).deposit{value: amountETH}();
112        assert(IWETH(WETH).transfer(pair, amountETH));
113        liquidity = IOxDexPair(pair).mint(to);
114        // refund dust eth, if any
115        if (msg.value > amountETH) TransferHelper.safeTransferETH(msg.sender, msg.value - amountETH);
116    }
117 }
```

图 22 addLiquidityETH 函数源码

- 相关函数: addLiquidity、addLiquidityETH、\_addLiquidity
- 安全建议: 无
- 审计结果: 通过

### 3.5.2 移除流动性

- 业务描述:

持有对应交易对 LP 代币的用户可调用 removeLiquidity、removeLiquidityETH、removeLiquidityWithPermit、removeLiquidityETHWithPermit、removeLiquidityETHSupportingFeeOnTransferTokens 和 removeLiquidityETHWithPermitSupportingFeeOnTransferTokens 函数来减少指定交易对的流动性。下如为 removeLiquidity 函数源码，该函数会先根据代币地址获取对应的交易对地址，然后将对应的 LP 代币发送交易对，并调用对应交易对的 burn 函数减少流动性。

```
126     function removeLiquidity(
127         address tokenA,
128         address tokenB,
129         uint256 liquidity,
130         uint256 amountAMin,
131         uint256 amountBMin,
132         address to,
133         uint256 deadline
134     ) public virtual override ensure(deadline) returns (uint256 amountA, uint256 amountB) {
135         address pair = OxDexLibrary.pairFor(factory, tokenA, tokenB);
136         IOxDexPair(pair).transferFrom(msg.sender, pair, liquidity); // send liquidity to pair
137         (uint256 amount0, uint256 amount1) = IOxDexPair(pair).burn(to);
138         (address token0, ) = OxDexLibrary.sortTokens(tokenA, tokenB);
139         (amountA, amountB) = tokenA == token0 ? (amount0, amount1) : (amount1, amount0);
140         require(amountA >= amountAMin, 'OxDexRouter: INSUFFICIENT_A_AMOUNT');
141         require(amountB >= amountBMin, 'OxDexRouter: INSUFFICIENT_B_AMOUNT');
142     }
```

图 23 removeLiquidity 函数源码

- **相关函数:** removeLiquidity、removeLiquidityETH、removeLiquidityWithPermit、removeLiquidityETHWithPermit、removeLiquidityETHSupportingFeeOnTransferTokens、removeLiquidityETHWithPermitSupportingFeeOnTransferTokens
- **安全建议:** 无
- **审计结果:** 通过

### 3.5.3 代币兑换

#### ➤ 业务描述:

持有交易对代币的用户可调用合约中多个兑换函数进行不同类型的代币兑换，分别是：以确定输入数量的代币兑换另一种代币(swapExactTokensForTokens)、以确定输出数量的代币兑换另一种代币(swapTokensForExactTokens)、以确定输入数量的ETH兑换代币(swapExactETHForTokens)、以确定输出数量的ETH代币(swapTokensForExactETH)、以确定输入数量的代币兑换ETH(swapExactTokensForETH)、以ETH兑换确定输出数量的代币(swapETHForExactTokens)、为获取对应支持手续费代币而兑换代币(swapExactTokensForTokensSupportingFeeOnTransferTokens)、以确定数量的ETH兑换支持手续费的代币(swapExactETHForTokensSupportingFeeOnTransferTokens)、以确定数量的代币兑换为支持手续费的ETH(swapExactTokensForETHSupportingFeeOnTransferTokens)。

这几种兑换方式的基本逻辑相似，以 swapExactETHForTokens 函数为例，如下图所示，兑换主要分为三步：首先，通过对对应的兑换数量计算函数(\_getAmountsOut)计算出本次兑换所需和可兑换出的代币数量；然后，将本次兑换所需的代币发送至对应的交易对合约(如果是 ETH，则需要本合约将 ETH 兑换为 WETH，再又本合约发送至对应的交易对地址)；最后是将本次兑换的所需的手续费发送至交易对合约地址，并缴纳手续费。



```
297     function swapExactETHForTokens(
298         uint256 amountOutMin,
299         address[] calldata path,
300         address to,
301         uint256 deadline
302     ) external payable virtual override ensure(deadline) returns (uint256[] memory amounts) {
303         require(path[0] == WETH, '0xDexRouter: INVALID_PATH');
304         amounts = OxDexLibrary.getAmountsOut(factory, msg.value, path);
305         require(amounts[amounts.length - 1] >= amountOutMin, '0xDexRouter: INSUFFICIENT_OUTPUT_AMOUNT');
306         IWETH(WETH).deposit{value: amounts[0]}();
307         assert(IWETH(WETH).transfer(OxDexLibrary.pairFor(factory, path[0], path[1]), amounts[0]));
308         _swap(amounts, path, to);
309     }
```

图 24 swapExactETHForTokens 函数源码

- **相关函数:** swapExactTokensForTokens、swapTokensForExactTokens、  
swapExactETHForTokens、swapTokensForExactETH、swapExactTokensForETH、  
swapETHForExactTokens、swapExactTokensForTokensSupportingFeeOnTransferTokens、  
swapExactETHForTokensSupportingFeeOnTransferTokens、  
swapExactTokensForETHSupportingFeeOnTransferTokens
- **安全建议:** 无
- **审计结果:** 通过

## 4 结论

Beosin(成都链安)对 0xDex 项目下智能合约的设计和代码实现进行了详细的审计。所有在审计过程中发现的问题均已书写至本审计报告，并通知项目方已进行修复。另需说明：(1)StakingRewards 合约中的管理员可设置当前周期的结束时间，如果输入某些特定值可能造成合约执行异常，无法正常运行；(2)StakingRewards 合约中的管理员可提取合约的奖励代币，(3)StakingRewards 合约的暂停功能并不影响 stakeWithPermit 函数。经项目方确定，这几个问题均是为了满足项目的设计需求，承诺会谨慎操作。因此，项目 0xDex 的智能合约的总体审计结果仍是通过。



成都链安  
BEOSIN

官方网址

<https://lianantech.com>

电子邮箱

vaas@lianantech.com

微信公众号

